h11 Documentation

Release 0.5.0

Nathaniel J. Smith

May 14, 2016

Contents

h11 is an HTTP/1.1 protocol library written in Python, heavily inspired by hyper-h2.

h11's goal is to be a simple, robust, complete, and non-hacky implementation of the first "chapter" of the HTTP/1.1 spec: RFC 7230: HTTP/1.1 Message Syntax and Routing. That is, it mostly focuses on implementing HTTP at the level of taking bytes on and off the wire, and the headers related to that, and tries to be picky about spec conformance when possible. It doesn't know about higher-level concerns like URL routing, conditional GETs, cross-origin cookie policies, or content negotiation. But it does know how to take care of framing, cross-version differences in keep-alive handling, and the "obsolete line folding" rule, and to use bounded time and space to process even pathological / malicious input, so that you can focus your energies on the hard / interesting parts for your application. And it tries to support the full specification in the sense that any useful HTTP/1.1 conformant application should be able to use h11.

This is a "bring-your-own-I/O" protocol library; like h2, it contains no IO code whatsoever. This means you can hook h11 up to your favorite network API, and that could be anything you want: synchronous, threaded, asynchronous, or your own implementation of RFC 6214 – h11 won't judge you. This is h11's main feature compared to the current state of the art, where every HTTP library is tightly bound to a particular network framework, and every time a new network API comes along then someone has to start over reimplementing the entire HTTP stack from scratch. We highly recommend Cory Benfield's excellent blog post about the advantages of this approach.

This also means that h11 is not immediately useful out of the box: it's a toolkit for building programs that speak HTTP, not something that could directly replace `requests` or `twisted.web` or whatever. But h11 makes it much easier to implement something like `requests` or `twisted.web`.

Vital statistics:

- Requirements: Python 2.7 or Python 3.3+, including PyPy
- Install: *not yet*
- Source: https://github.com/njsmith/h11
- Docs: https://h11.readthedocs.io
- License: MIT
- Code of conduct: Contributors are requested to follow our code of conduct in all project spaces.

CHAPTER 1

Contents

1.1 Getting started: Writing your own HTTP/1.1 client

h11 can be used to implement both HTTP/1.1 clients and servers. To give a flavor for how the API works, we'll demonstrate a small client.

1.1.1 HTTP basics

An HTTP interaction always starts with a client sending a *request*, optionally some *data* (e.g., a POST body); and then the server responds with a *response* and optionally some *data* (e.g. the requested document). Requests and responses have some data associated with them: for requests, this is a method (e.g. `GET`), a target (e.g. `/index.html`), and a collection of headers (e.g. `User-agent: demo-clent`). For responses, it's a status code (e.g. 404 Not Found) and a collection of headers.

Of course, as far as the network is concerned, there's no such thing as "requests" and "responses" – there's just bytes being sent from one computer to another. Let's see what this looks like, by fetching https://httpbin.org/xml:

```
In [1]: import ssl, socket

In [2]: ctx = ssl.create_default_context()

In [3]: sock = ctx.wrap_socket(socket.create_connection(("httpbin.org", 443)),
   ...:                        server_hostname="httpbin.org")
   ...:

# Send request
In [4]: sock.sendall(b"GET /xml HTTP/1.1\r\nhost: httpbin.org\r\n\r\n")
Out[4]: 40

# Read response
In [5]: response_data = sock.recv(1024)

# Let's see what we got!
In [6]: print(response_data)
b'HTTP/1.1 200 OK\r\nServer: nginx\r\nDate: Sat, 14 May 2016 01:32:22 GMT\r\nContent-Type: applicatio
```

So that's, uh, very convenient and readable. It's a little more understandable if we print the bytes as text:

```
In [7]: print(response_data.decode("ascii"))
HTTP/1.1 200 OK
Server: nginx
```

```
Date: Sat, 14 May 2016 01:32:22 GMT
Content-Type: application/xml
Content-Length: 522
Connection: keep-alive
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

<?xml version='1.0' encoding='us-ascii'?>

<!--  A SAMPLE set of slides  -->

<slideshow
    title="Sample Slide Show"
    date="Date of publication"
    author="Yours Truly"
    >

    <!-- TITLE SLIDE -->
    <slide type="all">
      <title>Wake up to WonderWidgets!</title>
    </slide>

    <!-- OVERVIEW -->
    <slide type="all">
        <title>Overview</title>
        <item>Why <em>WonderWidgets</em> are great</item>
        <item/>
        <item>Who <em>buys</em> WonderWidgets</item>
    </slide>

</slideshow>
```

Here we can see the status code at the top (200, which is the code for "OK"), followed by the headers, followed by the data (a silly little XML document). But we can already see that working with bytes by hand like this is really cumbersome. What we need to do is to move up to a higher level of abstraction.

This is what h11 does. Instead of talking in bytes, it lets you talk in high-level HTTP "events". To see what this means, let's repeat the above exercise, but using h11. We start by making a TLS connection like before, but now we'll also import *h11*, and create a *h11.Connection* object:

```
In [8]: import ssl, socket

In [9]: import h11

In [10]: ctx = ssl.create_default_context()

In [11]: sock = ctx.wrap_socket(socket.create_connection(("httpbin.org", 443)),
   ....:                        server_hostname="httpbin.org")
   ....:

In [12]: conn = h11.Connection(our_role=h11.CLIENT)
```

Next, to send an event to the server, there are three steps we have to take. First, we create an object representing the event we want to send – in this case, a *h11.Request*:

```
In [13]: request = h11.Request(method="GET",
   ....:                       target="/xml",
   ....:                       headers=[("Host", "httpbin.org")])
   ....:
```

Next, we pass this to our connection's *send()* method, which gives us back the bytes corresponding to this message:

```
In [14]: bytes_to_send = conn.send(request)
```

And then we send these bytes across the network:

```
In [15]: sock.sendall(bytes_to_send)
Out[15]: 40
```

There's nothing magical here – these are the same bytes that we sent up above:

```
In [16]: bytes_to_send
Out[16]: b'GET /xml HTTP/1.1\r\nhost: httpbin.org\r\n\r\n'
```

Why doesn't h11 go ahead and send the bytes for you? Because it's designed to be usable no matter what socket API you're using – doesn't matter if it's synchronous like this, asynchronous, callback-based, whatever; if you can read and write bytes from the network, then you can use h11.

In this case, we're not quite done yet – we have to send another event to tell the other side that we're finished, which we do by sending an *EndOfMessage* event:

```
In [17]: end_of_message_bytes_to_send = conn.send(h11.EndOfMessage())

In [18]: sock.sendall(end_of_message_bytes_to_send)
Out[18]: 0
```

Of course, it turns out that in this case, the HTTP/1.1 specification tells us that any request that doesn't contain either a `Content-Length` or `Transfer-Encoding` header automatically has a 0 length body, and h11 knows that, and h11 knows that the server knows that, so it actually encoded the *EndOfMessage* event as the empty string:

```
In [19]: end_of_message_bytes_to_send
Out[19]: b''
```

But there are other cases where it might not, depending on what headers are set, what message is being responded to, the HTTP version of the remote peer, etc. etc. So for consistency, h11 requires that you *always* finish your messages by sending an explicit *EndOfMessage* event; then it keeps track of the details of what that actually means in any given situation, so that you don't have to.

Finally, we have to read the server's reply. By now you can probably guess how this is done: we read some bytes from the network, then we hand them to *Connection.receive_data()* and it gives us back high-level events from the server.

```
In [20]: bytes_received = sock.recv(1024)

In [21]: events_received = conn.receive_data(bytes_received)

In [22]: events_received
Out[22]:
[Response(status_code=200, headers=[(b'server', b'nginx'), (b'date', b'Sat, 14 May 2016 01:32:22 GMT
 Data(data=bytearray(b'<?xml version=\'1.0\' encoding=\'us-ascii\'?>\n\n<!--  A SAMPLE set of slides
 EndOfMessage(headers=[])]
```

Here the server sent us three events: a *Response* object, which is similar to the *Request* object that we created earlier and has the response's status code (200 OK) and headers; a *Data* object containing the response data; and another *EndOfMessage* object. This similarity between what we send and what we receive isn't accidental: if we were using h11 to write an HTTP server, then these are the objects we would have created and passed to *send()* – h11 in client and server mode has an API that's almost exactly symmetric.

1.1.2 A basic client object

To make this a little more convenient to play with, we can wrap up our socket and *Connection* into a single object with some convenience methods:

```
import socket, ssl
import h11

class MyHttpClient:
    def __init__(self, host, port):
        self.sock = socket.create_connection((host, port))
        if port == 443:
            self.sock = ssl.wrap_socket(self.sock)
        self.conn = h11.Connection(our_role=h11.CLIENT)

    def send(self, *events):
        for event in events:
            data = self.conn.send(event)
            if data is None:
                # event was a ConnectionClosed(), meaning that we won't be
                # sending any more data:
                self.sock.shutdown(socket.SHUT_WR)
            else:
                self.sock.sendall(data)

    # max_bytes set intentionally small for pedagogical purposes
    def receive(self, max_bytes=200):
        return self.conn.receive_data(self.sock.recv(max_bytes))
```

And then we can send requests:

```
In [23]: client = MyHttpClient("httpbin.org", 443)

In [24]: client.send(h11.Request(method="GET", target="/xml",
   ....:                         headers=[("Host", "httpbin.org")]),
   ....:             h11.EndOfMessage())
   ....:
```

And read back the events:

```
In [25]: client.receive()
Out[25]: []
```

What happened here? We only read a max of 200 bytes from the socket (see `max_bytes=` above), and it turns out that this wasn't enough to form a complete event. This happens all the time in real life, due to slow networks or whatever – data trickles in at its own pace. When this happens, h11 buffers the unprocessed data internally, and if you keep reading then eventually you'll get a complete event:

```
In [26]: client.receive()
Out[26]:
[Response(status_code=200, headers=[(b'server', b'nginx'), (b'date', b'Sat, 14 May 2016 01:32:23 GMT
 Data(data=bytearray(b'<?xml version=\'1.0\' encoding=\'us-ascii\'?>\n\n<!--  A SAMPLE set of slides
```

Note here that we received a *Data* event that only has *part* of the response body – h11 streams out data as it arrives, which might mean that you receive multiple *Data* events. (Of course, if you're the one sending data, you can do the same thing: instead of buffering all your data in one giant *Data* event, you can send multiple *Data* events yourself to stream the data out incrementally; just make sure that you set the appropriate `Content-Length` / `Transfer-Encoding` headers.) If we keep reading, we'll see more *Data* events, and then eventually the *EndOfMessage*:

```
In [27]: client.receive()
Out[27]: [Data(data=bytearray(b'\n\n    <!-- TITLE SLIDE -->\n    <slide type="all">\n      <title>Wa

In [28]: client.receive()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
[Data(data=bytearray(b'm>Why <em>WonderWidgets</em> are great</item>\n        <item/>\n        <item>
 EndOfMessage(headers=[])]
```

Now we can see why *EndOfMessage* is so important – otherwise, we can't tell when we've received the end of the data. And since that's the end of this response, the server won't send us anything more until we make another request – if we try, then the socket read will just hang forever, unless we set a timeout or interrupt it:

```
In [29]: client.sock.settimeout(2)

In [30]: client.receive()

timeoutTraceback (most recent call last)
<ipython-input-30-4394c01d9ea0> in <module>()
----> 1 client.receive()

<string> in receive(self, max_bytes)

/usr/lib/python3.4/ssl.py in recv(self, buflen, flags)
    752                     "non-zero flags not allowed in calls to recv() on %s" %
    753                     self.__class__)
--> 754             return self.read(buflen)
    755         else:
    756             return socket.recv(self, buflen, flags)

/usr/lib/python3.4/ssl.py in read(self, len, buffer)
    641                 v = self._sslobj.read(len, buffer)
    642             else:
--> 643                 v = self._sslobj.read(len or 1024)
    644             return v
    645         except SSLError as x:

timeout: The read operation timed out
```

1.1.3 Keep-alive

For some servers, we'd have to stop here, because they require a new connection for every request/response. But, this server is smarter than that – it supports keep-alive, so we can re-use this connection to send another request. There's a few ways we can tell. First, if it didn't, then it would have closed the connection already, and we would have gotten a *ConnectionClosed* event on our last call to `receive()`. We can also tell by checking h11's internal idea of what state the two sides of the conversation are in:

```
In [31]: client.conn.our_state, client.conn.their_state
Out[31]: (DONE, DONE)
```

If the server didn't support keep-alive, then these would be *MUST_CLOSE* and either *MUST_CLOSE* or *CLOSED*, respectively (depending on whether we'd seen the socket actually close yet). *DONE* / *DONE*, on the other hand, means that this request/response cycle has totally finished, but the connection itself is still viable, and we can start over and send a new request on this same connection.

To do this, we tell h11 to get ready (this is needed as a safety measure to make sure different requests/responses on the same connection don't get accidentally mixed up):

```
In [32]: client.conn.prepare_to_reuse()
```

This resets both sides back to their initial *IDLE* state, allowing us to send another *Request*:

```
In [33]: client.conn.our_state, client.conn.their_state
Out[33]: (IDLE, IDLE)

In [34]: client.send(h11.Request(method="GET", target="/get",
   ....:                         headers=[("Host", "httpbin.org")]),
   ....:             h11.EndOfMessage())
   ....:

In [35]: client.receive(max_bytes=4096)
Out[35]:
[Response(status_code=200, headers=[(b'server', b'nginx'), (b'date', b'Sat, 14 May 2016 01:32:25 GMT
 Data(data=bytearray(b'{\n  "args": {}, \n  "headers": {\n    "Host": "httpbin.org"\n  }, \n  "origi
 EndOfMessage(headers=[])]
```

1.1.4 What's next?

Here's some ideas of things you might try:

- Adapt the above examples to make a POST request. (Don't forget to set the `Content-Length` header – but don't worry, if you do forget, then h11 will give you an error when you try to send data):

```
client.send(h11.Request(method="POST", target="/post",
                        headers=[("Host", "httpbin.org"),
                                 ("Content-Length", "10")]),
            h11.Data(data=b"1234567890"),
            h11.EndOfMessage())
client.receive(max_bytes=4096)
```

- Experiment with what happens if you try to violate the HTTP protocol by sending a *Response* as a client, or sending two *Request*s in a row.
- Write your own basic `http_get` function that takes a URL, parses out the host/port/path, then connects to the server, does a `GET` request, and then collects up all the resulting *Data* objects, concatenates their payloads, and returns it.
- Adapt the above code to use your favorite non-blocking API
- Use h11 to write a simple HTTP server. (If you get stuck, there's an example in the test suite.)

And of course, you'll want to read the *API documentation* for all the details.

1.2 API documentation

Contents

h11 has a fairly small public API, with all public symbols available directly at the top level:

```
In [1]: import h11

In [2]: h11.<TAB>
h11.CLIENT                  h11.MUST_CLOSE
h11.CLOSED                  h11.Paused
h11.Connection              h11.PRODUCT_ID
h11.ConnectionClosed        h11.ProtocolError
h11.Data                    h11.Request
h11.DONE                    h11.Response
h11.EndOfMessage            h11.SEND_BODY
h11.ERROR                   h11.SEND_RESPONSE
h11.IDLE                    h11.SERVER
h11.InformationalResponse   h11.SWITCHED_PROTOCOL
h11.MIGHT_SWITCH_PROTOCOL
```

These symbols fall into three main categories: event classes, special constants used to track different connection states, and the `Connection` class itself. We'll describe them in that order.

1.2.1 Events

Events are the core of h11: the whole point of h11 is to let you think about HTTP transactions as being a series of events sent back and forth between a client and a server, instead of thinking in terms of bytes.

All events behave in essentially similar ways. Let's take `Request` as an example. Like all events, this is a "final" class – you cannot subclass it. And like all events, it has several fields. For `Request`, there are four of them: `method`, `target`, `headers`, and `http_version`. `http_version` defaults to `b"1.1"`; the rest have no default, so to create a `Request` you have to specify their values:

```
In [3]: req = h11.Request(method="GET",
   ...:                   target="/",
   ...:                   headers=[("Host", "example.com")])
   ...:
```

Event constructors accept only keyword arguments, not positional arguments.

Events have a useful repr:

```
In [4]: req
Out[4]: Request(method=b'GET', target=b'/', headers=[(b'host', b'example.com')], http_version=b'1.1')
```

And their fields are available as regular attributes:

```
In [5]: req.method
Out[5]: b'GET'

In [6]: req.target
\\\\\\\\\\\\\\\Out[6]: b'/'

In [7]: req.headers
\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[7]: [(b'host', b'example.com')]

In [8]: req.http_version
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[8]: b'1.1'
```

Notice that these attributes have been normalized to byte-strings. In general, events normalize and validate their fields when they're constructed. Some of these normalizations and checks are specific to a particular event – for example, `Request` enforces RFC 7230's requirement that HTTP/1.1 requests must always contain a `"Host"` header:

```
# HTTP/1.0 requests don't require a Host: header
In [9]: h11.Request(method="GET", target="/", headers=[], http_version="1.0")
Out[9]: Request(method=b'GET', target=b'/', headers=[], http_version=b'1.0')
```

```
# But HTTP/1.1 requests do
In [10]: h11.Request(method="GET", target="/", headers=[])

ProtocolErrorTraceback (most recent call last)
<ipython-input-10-645a3ef89e11> in <module>()
----> 1 h11.Request(method="GET", target="/", headers=[])

/home/docs/checkouts/readthedocs.org/user_builds/h11/envs/v0.5.0/lib/python3.4/site-packages/h11-0.5
     54                 raise ProtocolError("status code must be integer")
     55
---> 56         self._validate()
     57
     58     def _validate(self):

/home/docs/checkouts/readthedocs.org/user_builds/h11/envs/v0.5.0/lib/python3.4/site-packages/h11-0.5
    121                     break
    122             else:
--> 123                 raise ProtocolError("Missing mandatory Host: header")
    124
    125

ProtocolError: Missing mandatory Host: header
```

This helps protect you from accidentally violating the protocol, and also helps protect you from remote peers who attempt to violate the protocol.

A few of these normalization rules are standard across multiple events, so we document them here: `headers`: In h11, headers are represented internally as a list of (*name*, *value*) pairs, where *name* and *value* are both byte-strings, *name* is always lowercase, and *name* and *value* are both guaranteed not to have any leading or trailing whitespace. When constructing an event, we accept any iterable of pairs like this, and will automatically convert native strings containing ascii or bytes-like objects to byte-strings, convert names to lowercase, and strip whitespace from values:

```
In [11]: original_headers = [("HOST", bytearray(b"  example.com   "))]

In [12]: req = h11.Request(method="GET", target="/", headers=original_headers)

In [13]: original_headers
```

```
Out[13]: [('HOST', bytearray(b'  example.com  '))]

In [14]: req.headers
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[14]: [(b'host', b'example.com')]
```

If any names are detected with leading or trailing whitespace, then this is an error ("in the past, differences in the handling of such whitespace have led to security vulnerabilities" – RFC 7230). We also check for other protocol violations, e.g. `Content-Length:  hello` is an error. We may add additional checks in the future. It's not just headers we normalize to being byte-strings: the same type-conversion logic is also applied to the *`Request.method`* and *`Request.target`* field, and – for consistency – all `http_version` fields. In particular, we always represent HTTP version numbers as byte-strings like `b"1.1"`. Bytes-like objects and native strings will be automatically converted to byte strings. Note that the HTTP standard specifically guarantees that all HTTP version numbers will consist of exactly two digits separated by a dot, so comparisons like `req.http_version < b"1.1"` are safe and valid.

When manually constructing an event, you generally shouldn't specify `http_version`, because it defaults to `b"1.1"`, and if you attempt to override this to some other value then *`Connection.send()`* will reject your event – h11 only speaks HTTP/1.1. But it does understand other versions of HTTP, so you might receive events with other `http_version` values from remote peers.

Here's the complete set of events supported by h11:

class `h11.`**`Request`**`(`***kwargs*`)`
The beginning of an HTTP request.

Fields:

`method`
An HTTP method, e.g. `b"GET"` or `b"POST"`. Always a byte string. Bytes-like objects and native strings containing only ascii characters will be automatically converted to byte strings.

`target`
The target of an HTTP request, e.g. `b"/index.html"`, or one of the more exotic formats described in RFC 7320, section 5.3. Always a byte string. Bytes-like objects and native strings containing only ascii characters will be automatically converted to byte strings.

`headers`
Request headers, represented as a list of (name, value) pairs. See *the header normalization rules* for details.

`http_version`
The HTTP protocol version, represented as a byte string like `b"1.1"`. See *the HTTP version normalization rules* for details.

class `h11.`**`InformationalResponse`**`(`***kwargs*`)`
An HTTP informational response.

Fields:

`status_code`
The status code of this response, as an integer. For an *`InformationalResponse`*, this is always in the range [100, 200).

`headers`
Request headers, represented as a list of (name, value) pairs. See *the header normalization rules* for details.

`http_version`
The HTTP protocol version, represented as a byte string like `b"1.1"`. See *the HTTP version normalization rules* for details.

class `h11.`**`Response`**`(`***kwargs*`)`
The beginning of an HTTP response.

Fields:

status_code
The status code of this response, as an integer. For an *Response*, this is always in the range [200, 600).

headers
Request headers, represented as a list of (name, value) pairs. See *the header normalization rules* for details.

http_version
The HTTP protocol version, represented as a byte string like `b"1.1"`. See *the HTTP version normalization rules* for details.

class h11.**Data**(***kwargs*)
Part of an HTTP message body.

Fields:

class h11.**EndOfMessage**(***kwargs*)
The end of an HTTP message.

Fields:

class h11.**ConnectionClosed**(***kwargs*)
This event indicates that the sender has closed their outgoing connection.

Note that this does not necessarily mean that they can't *receive* further data, because TCP connections are composed to two one-way channels which can be closed independently. See *Closing connections* for details.

No fields.

class h11.**Paused**(***kwargs*)
A pseudo-event used for flow control.

If *Connection.receive_data()* returns this event, it means that the HTTP parser is in a paused condition, and won't process any new data until after the condition is resolved. See *Flow control* for details.

reason
The remote peer's state that triggered the pause. One of:

- *h11.DONE*: a client has started sending another request before we finished responding to their first request. Cleared by finishing the response and then calling *Connection.prepare_to_reuse()*.
- *MIGHT_SWITCH_PROTOCOL*: a client is in the *MIGHT_SWITCH_PROTOCOL* state, and is waiting for the server to either accept or reject the proposed protocol switch. See *Switching protocols* for details.
- *SWITCHED_PROTOCOL*: the remote peer is the *SWITCHED_PROTOCOL* state. h11 isn't going to parse any more data that they send, because they're no longer speaking HTTP. See *Switching protocols* for details.

1.2.2 The state machine

Now that you know what the different events are, the next question is: what can you do with them?

A basic HTTP request/response cycle looks like this:

- The client sends:
  - one *Request* event with request metadata and headers,
  - zero or more *Data* events with the request body (if any),
  - and an *EndOfMessage* event.

- And then the server replies with:
  - zero or more *InformationalResponse* events,
  - one *Response* event,
  - zero or more *Data* events with the response body (if any),
  - and a *EndOfMessage* event.

And once that's finished, both sides either close the connection, or they go back to the top and re-use it for another request/response cycle.

To coordinate this interaction, the h11 *Connection* object maintains several state machines: one that tracks what the client is doing, one that tracks what the server is doing, and a few more tiny ones to track whether *keep-alive* is enabled and whether the client has proposed to *switch protocols*. h11 always keeps track of all of these state machines, regardless of whether it's currently playing the client or server role.

The state machines look like this (click on each to expand):

If you squint at the first two diagrams, you can see the client's IDLE -> SEND_BODY -> DONE path and the server's IDLE -> SEND_RESPONSE -> SEND_BODY -> DONE path, which encode the basic sequence of events we described above. But there's a fair amount of other stuff going on here as well.

The first thing you should notice is the different colors. These correspond to the different ways that our state machines can change state.

- Dark blue arcs are *event-triggered transitions*: if we're in state A, and this event happens, when we switch to state B. For the client machines, these transitions always happen when the client *sends* an event. For the server machine, most of them involve the server sending an event, except that the server also goes from IDLE -> SEND_RESPONSE when the client sends a *Request*.
- Green arcs are *state-triggered transitions*: these are somewhat unusual, and are used to couple together the different state machines – if, at any moment, one machine is in state A and another machine is in state B, then the first machine immediately transitions to state C. For example, if the CLIENT machine is in state DONE, and the SERVER machine is in the CLOSED state, then the CLIENT machine transitions to MUST_CLOSE. And the same thing happens if the CLIENT machine is in the state DONE and the keep-alive machine is in the state disabled.
- There are also two purple arcs labeled `prepare_to_send()`: these correspond to an explicit method call documented below.

Here's why we have all the stuff in those diagrams above, beyond what's needed to handle the basic request/response cycle:

- Server sending a *Response* directly from *IDLE*: This is used for error responses, when the client's request never arrived (e.g. 408 Request Timed Out) or was unparseable gibberish (400 Bad Request) and thus didn't register with our state machine as a real *Request*.
- The transitions involving *MUST_CLOSE* and `CLOSE`: keep-alive and shutdown handling; see *Re-using a connection: keep-alive and pipelining* and *Closing connections*.
- The transitions involving *MIGHT_SWITCH_PROTOCOL* and *SWITCHED_PROTOCOL*: See *Switching protocols*.
- That weird *ERROR* state hanging out all lonely on the bottom: to avoid cluttering the diagram, we don't draw any arcs coming into this node, but that doesn't mean it can't be entered. In fact, it can be entered from any state: if any exception occurs while trying to send/receive data, then the corresponding machine will transition directly to this state. Once there, though, it can never leave – that part of the diagram is accurate. See *Error handling*.

And finally, note that in these diagrams, all the labels that are in *italics* are informal English descriptions of things that happen in the code, while the labels in upright text correspond to actual objects in the public API. You've already seen the event objects like *Request* and *Response*; there are also a set of opaque sentinel values that you can use to track and query the client and server's states:

h11.**IDLE**

h11.**SEND_RESPONSE**

h11.**SEND_BODY**

h11.**DONE**

h11.**MUST_CLOSE**

h11.**CLOSED**

h11.**MIGHT_SWITCH_PROTOCOL**

h11.**SWITCHED_PROTOCOL**

h11.**ERROR**

For example, we can see that initially the client and server start in state *IDLE* / *IDLE*:

```
In [15]: conn = h11.Connection(our_role=h11.CLIENT)

In [16]: conn.states
Out[16]: {SERVER: IDLE, CLIENT: IDLE}
```

And then if the client sends a *Request*, then the client switches to state *SEND_BODY*, while the server switches to state *SEND_RESPONSE*:

```
In [17]: conn.send(h11.Request(method="GET", target="/", headers=[("Host", "example.com")]));

In [18]: conn.states
Out[18]: {SERVER: SEND_RESPONSE, CLIENT: SEND_BODY}
```

And we can test these values directly using constants like *SEND_BODY*:

```
In [19]: conn.states[h11.CLIENT] is h11.SEND_BODY
Out[19]: True
```

This shows how the *Connection* type tracks these state machines and lets you query their current state.

1.2.3 The Connection object

There are two special constants used to indicate the two different roles that a peer can play in an HTTP connection:

h11.**CLIENT**

h11.**SERVER**

When creating a *Connection* object, you need to pass one of these constants to indicate which side of the HTTP conversation you want to implement:

class h11.**Connection**(*our_role*, *max_buffer_size=16384*)

An object encapsulating the state of an HTTP connection.

Parameters

- **our_role** – If you're implementing a client, pass *h11.CLIENT*. If you're implementing a server, pass *h11.SERVER*.

- **max_buffer_size** (*int*) – The maximum number of bytes of received but unprocessed data we're willing to buffer. In practice this mostly sets a limit on the maximum size of the request/response line + headers. If this is exceeded, then *receive_data()* will raise *ProtocolError*.

receive_data(*data*)

Convert bytes received from the remote peer into high-level events, while updating our internal state machine.

Parameters **data** (bytes-like object, or None) – The new data that was just recieved.

Normally, *data* is a bytes-like object containing new data received from the peer. We append this to our internal receive buffer, and then check whether any new events can be parsed from it. We always parse and return as many events as possible.

There are two important special cases:

Special case 1: If *data* is an empty byte-string like `b""`, then this indicates that the remote side has closed the connection (end of file). Normally this is convenient, because standard Python APIs like `file.read()` or `socket.recv()` use `b""` to indicate end-of-file, while other failures to read are indicated using other mechanisms like raising `TimeoutError`. When using such an API you can just blindly pass through whatever you get from `read` to *receive_data()*, and everything will work.

But, if you have an API where reading an empty string is a valid non-EOF condition, then you need to be aware of this and make sure to check for such strings and avoid passing them to *receive_data()*.

Special case 2: If *data* is `None`, then we don't add any data to the internal receive buffer, but we attempt to parse it again to see if we can pull any new events out.

receive_data() normally pulls out all possible events immediately, so this is only useful after calling *prepare_to_reuse()* – see *Re-using a connection: keep-alive and pipelining* for details.

Returns A list of *event* objects.

Raises *ProtocolError*

The peer has misbehaved. You should close the connection (possibly after sending some kind of 400 response).

For robustness you might want to be prepared to catch other exceptions as well, but if this happens then please do file a bug report as well – the intention is that *ProtocolError* is the *only* exception that this method should be able to raise.

If this method raises any exception then it also sets *Connection.their_state* to *ERROR* – see *Error handling* for discussion.

send(*event*)

Convert a high-level event into bytes that can be sent to the peer, while updating our internal state machine.

Parameters **event** – The *event* to send.

Returns If `type(event) is ConnectionClosed`, then returns `None`. Otherwise, returns a bytes-like object.

Raises *ProtocolError*

Sending this event at this time would violate our understanding of the HTTP/1.1 protocol.

If this method raises any exception then it also sets *Connection.our_state* to *ERROR* – see *Error handling* for discussion.

send_with_data_passthrough(*event*)
Identical to *send()*, except that in situations where *send()* returns a single bytes-like object, this instead returns a list of them – and when sending a *Data* event, this list is guaranteed to contain the exact object you passed in as `Data.data`. See *Support for sendfile()* for discussion.

prepare_to_reuse()
Attempt to reset our connection state for a new request/response cycle.

If both client and server are in *DONE* state, then resets them both to *IDLE* state in preparation for a new request/response cycle on this same connection. Otherwise, raises a *ProtocolError*.

See *Re-using a connection: keep-alive and pipelining*.

our_role
CLIENT if this is a client; *SERVER* if this is a server.

their_role
SERVER if this is a client; *CLIENT* if this is a server.

states
A dictionary like:

```
{CLIENT: <client state>, SERVER: <server state>}
```

See *The state machine* for details.

our_state
The current state of whichever role we are playing. See *The state machine* for details.

their_state
The current state of whichever role we are NOT playing. See *The state machine* for details.

their_http_version
The version of HTTP that our peer claims to support. `None` if we haven't yet received a request/response.

This is preserved by *prepare_to_reuse()*, so it can be handy for a client making multiple requests on the same connection: normally you don't know what version of HTTP the server supports until after you do a request and get a response – so on an initial request you might have to assume the worst. But on later requests on the same connection, the information will be available here.

client_is_waiting_for_100_continue
True if the client sent a request with the `Expect: 100-continue` header, and is still waiting for a response (i.e., the server has not sent a 100 Continue or any other kind of response, and the client has not gone ahead and started sending the body anyway).

See RFC 7231 section 5.1.1 for details.

they_are_waiting_for_100_continue
True if *their_role* is *CLIENT* and *client_is_waiting_for_100_continue*.

trailing_data
Data that has been received, but not yet processed, represented as a tuple with two elements, where the first is a byte-string containing the unprocessed data itself, and the second is a bool that is True if the receive connection was closed.

See *Switching protocols* for discussion of why you'd want this.

1.2.4 Error handling

Given the vagaries of networks and the folks on the other side of them, it's extremely important to be prepared for errors.

Most errors in h11 are signaled by raising *ProtocolError*:

exception h11.**ProtocolError**(*msg*, *error_status_hint=400*)

This exception indicates a violation of the HTTP/1.1 protocol.

This might be because your perr tried to do something that HTTP/1.1 says is illegal (if it's raised by *Connection.receive_data()*), or that you did. Either way, you should probably close the connection and think things over.

In addition to the normal Exception features, it has one attribute:

error_status_hint

If you're a server and you want to send an error response back to a naughty client, then this gives a suggestion as to which status code you might want to use. The default is 400 Bad Request, a generic catch-all for protocol violations.

There are four cases where this exception might be raised:

- When trying to instantiate an event object: This indicates that something about your event is invalid. Your event wasn't constructed, but there are no other consequences – feel free to try again.
- When calling *Connection.prepare_to_reuse()*: This indicates that the connection is not ready to be re-used, because one or both of the peers are not in the *DONE* state. The *Connection* object remains usable, and you can try again later.
- When calling *Connection.receive_data()*: This indicates that the remote peer has violated our protocol assumptions. This is unrecoverable – we don't know what they're doing and we cannot safely proceed. *Connection.their_state* immediately becomes *ERROR*, and all further calls to *receive_data()* will also raise *ProtocolError*. *Connection.send()* still works as normal, so if you're implementing a server and this happens then you have an opportunity to send back a 400 Bad Request response. Your only other real option is to close your socket and make a new connection.
- When calling *Connection.send()*: This indicates that *you* violated our protocol assumptions. This is also unrecoverable – h11 doesn't know what you're doing, its internal state may be inconsistent, and we cannot safely proceed. *Connection.our_state* immediately becomes *ERROR*, and all further calls to *send()* will also raise *ProtocolError*. The only thing you can reasonably due at this point is to close your socket and make a new connection.

1.2.5 Message body framing: `Content-Length` and all that

There are two different headers that HTTP/1.1 uses to indicate a framing mechanism for request/response bodies: `Content-Length` and `Transfer-Encoding`. Our general philosophy is that the way you tell h11 what configuration you want to use is by setting the appropriate headers in your request / response, and then h11 will both pass those headers on to the peer and encode the body appropriately.

Currently, the only supported `Transfer-Encoding` is `chunked`.

On requests, this means:

- No `Content-Length` or `Transfer-Encoding`: no body, equivalent to `Content-Length: 0`.
- `Content-Length: ...`: You're going to send exactly the specified number of bytes. h11 will keep track and signal an error if your *EndOfMessage* doesn't happen at the right place.
- `Transfer-Encoding: chunked`: You're going to send a variable / not yet known number of bytes.

  Note 1: only HTTP/1.1 servers are required to supported `Transfer-Encoding: chunked`, and as a client you have to either send this header or not before you get to see what protocol version the server is using.

Note 2: even though HTTP/1.1 servers are required to support `Transfer-Encoding: chunked`, this doesn't mean that they actually do – e.g., applications using Python's standard WSGI API cannot accept chunked requests.

Nonetheless, this is the only way to send request where you don't know the size of the body ahead of time, so you might as well go ahead and hope.

On responses, things are a bit more subtle. There are effectively two cases:

- `Content-Length: ...`: You're going to send exactly the specified number of bytes. h11 will keep track and signal an error if your *EndOfMessage* doesn't happen at the right place.
- `Transfer-Encoding: chunked`, *or*, neither framing header is provided: These two cases are handled differently at the wire level, but as far as the application is concerned they provide (almost) exactly the same semantics: in either case, you'll send a variable / not yet known number of bytes. The difference between them is that `Transfer-Encoding: chunked` works better (compatible with keep-alive, allows trailing headers, clearly distinguishes between successful completion and network errors), but requires an HTTP/1.1 client; for HTTP/1.0 clients the only option is the no-headers close-socket-to-indicate-completion approach.

  Since this is (almost) entirely a wire-level-encoding concern, h11 abstracts it: when sending a response you can set either `Transfer-Encoding: chunked` or leave off both framing headers, and h11 will treat both cases identically: it will automatically pick the best option given the client's advertised HTTP protocol level.

  You need to watch out for this if you're using trailing headers (i.e., a non-empty `headers` attribute on *EndOfMessage*), since trailing headers are only legal if we actually ended up using `Transfer-Encoding: chunked`. Trying to send a non-empty set of trailing headers to a HTTP/1.0 client will raise a *ProtocolError*. If this use case is important to you, check *Connection.their_http_version* to confirm that the client speaks HTTP/1.1 before you attempt to send any trailing headers.

1.2.6 Re-using a connection: keep-alive and pipelining

HTTP/1.1 allows a connection to be re-used for multiple request/response cycles (also known as "keep-alive"). This can make things faster by letting us skip the costly connection setup, but it does create some complexities: we have to keep track of whether a connection is reusable, and when there are multiple requests and responses flowing through the same connection we need to be careful not to get confused about which request goes with which response.

h11 considers a connection to be reusable if, and only if, both sides (a) speak HTTP/1.1 (HTTP/1.0 did have some complex and fragile support for keep-alive bolted on, but h11 currently doesn't support that – possibly this will be added in the future), and (b) neither side has explicitly disabled keep-alive by sending a `Connection: close` header.

If you plan to make only a single request or response and then close the connection, you should manually set the `Connection: close` header in your request/response. h11 will notice and update its state appropriately.

There are also some situations where you are required to send a `Connection: close` header, e.g. if you are a server talking to a client that doesn't support keep-alive. You don't need to worry about these cases – h11 will automatically add this header when necessary. Just worry about setting it when it's actually something that you're actively choosing.

If you want to re-use a connection, you have to wait until both the request and the response have been completed, bringing both the client and server to the *DONE* state. Once this has happened, you can explicitly call *Connection.prepare_to_reuse()* to reset both sides back to the *IDLE* state. This makes sure that the client and server remain synched up.

If keep-alive is disabled for whatever reason – explicit headers, lack of protocol support, one of the sides just unilaterally closed the connection – then the state machines will skip past the *DONE* state directly to the *MUST_CLOSE* or *CLOSED* states. In this case, trying to call `prepare_to_use()` will raise an error, and the only thing you can legally do is to close this connection and make a new one.

HTTP/1.1 also allows for a more aggressive form of connection re-use, in which a client sends multiple requests in quick succession, and then waits for the responses to stream back in order ("pipelining"). This is generally considered to have been a bad idea, because it makes things like error recovery very complicated.

As a client, h11 does not support pipelining. This is enforced by the structure of the state machine: after sending one *Request*, you can't send another until after calling *prepare_to_reuse()*, and you can't call *prepare_to_reuse()* until the server has entered the *DONE* state, which requires reading the server's full response.

As a server, h11 provides the minimal support for pipelining required to comply with the HTTP/1.1 standard: if the client sends multiple pipelined requests, then we the first request until we reach the *DONE* state, and then *receive_data()* will pause and refuse to parse any more events until the response is completed and *prepare_to_reuse()* is called. See the next section for more details.

1.2.7 Flow control

h11 always does the absolute minimum of buffering that it can get away with: *send()* always returns the full data to send immediately, and `recieve_data()` always greedily parses and returns as many events as possible from its current buffer. So you can be sure that no data or events will suddenly appear and need processing, except when you call these methods. And presumably you know when you want to send things. But there is one thing you still need to know: you don't want to read data from the remote peer if it can't be processed (i.e., you want to apply backpressure and avoid building arbitrarily large buffers), and you definitely don't want to block waiting on data from the remote peer at the same time that it's blocked waiting for you, because that will cause a deadlock.

We assume that if you're implementing a client then you're clever enough not to sit around trying to read more data from the server when there's no response pending. But there are a few more subtle ways that reading in HTTP can go wrong, and h11 provides two ways to help you avoid these situations.

First, it keeps track of the *client's ''Expect: 100-continue'* status <https://tools.ietf.org/html/rfc7231#section-5.1.1>'_. you can read the spec for details, but basically the way this works if that sometimes clients will send a *Request* with an `Expect: 100-continue` header, and then they will stop there, before sending the body, until they see some response from the server (or possibly some timeout occurs). The server's response can be an *InformationalResponse* with status `100 Continue`, or anything really (e.g. a full *Response* with an error code). The crucial thing as a server, though, is that you should never block trying to read a request body if the client is blocked waiting for you to tell them to send the request body.

The simple way to avoid this is to make sure that before you block waiting to read data, always execute some code like:

```
if conn.they_are_waiting_for_100_continue:
    send(conn, h11.InformationalResponse(100, headers=[...]))
do_read(...)
```

The other mechanism h11 provides to help you manage read flow control is the *Paused* pseudo-event. Unlike other events, the *Paused* event doesn't contain information sent from the remote peer; if *receive_data()* returns one of these, it means that *receive_data()* has stopped processing its data buffer and isn't going to process any more until the remote peer's state (*Connection.their_state*) changes to something different.

There are three possible reasons to enter a paused state:

- The remote peer is in the *DONE* state, but sent more data, i.e., a client is attempting to *pipeline requests*. In the *DONE* state, *receive_data()* can return *ConnectionClosed* events, but if any actual data is received then it will pause, and stay that way until a successful call to *prepare_to_reuse()*.
- The remote client is in the *MIGHT_SWITCH_PROTOCOL* state (see *Switching protocols*). This really shouldn't happen, because they don't know yet whether the protocol switch will actually happen, but OTOH it certainly isn't correct for us to go ahead and parse the data they sent as if it were HTTP, when it might not be. So if this happens, we pause.

- The remote peer is in the `SWITCHED_PROTOCOL` state (see *Switching protocols*). We certainly aren't going to try to parse their data – it's not HTTP, or at least not HTTP directed at us. If this happens, we pause.

Once the connection has entered a paused state, then it's safe to keep calling `receive_data()` – it will just keep returning new `Paused` events – but instead you should probably stop reading from the network; all you're going to accomplish is to shove more and more data into our internal buffers, where it's just going to there using more and more memory. (And we do *not* enforce the regular maximum buffer size limits when in a paused state – if we did then you might go over the limit in a single call to `receive_data()`, not because you or the remote peer did anything wrong, but just because a fair amount of data all came in at the same time we entered the paused state.) And simply reading more data will never trigger an unpause – for that something external has to happen, usually a call to `prepare_to_reuse()`.

And that's the other tricky moment: when you come out of a paused state, you shouldn't immediately read from the network. Consider the situation where a client sends two pipelined requests, and then blocks waiting for the two responses. It's possible the two requests will arrive together, and be enqueued into our receive buffer together:

```
In [20]: conn = h11.Connection(our_role=h11.SERVER)

In [21]: conn.receive_data(
   ....:     b"GET /1 HTTP/1.1\r\nHost: example.com\r\n\r\n"
   ....:     b"GET /1 HTTP/1.1\r\nHost: example.com\r\n\r\n"
   ....: )
   ....:
Out[21]:
[Request(method=b'GET', target=b'/1', headers=[(b'host', b'example.com')], http_version=b'1.1'),
 EndOfMessage(headers=[]),
 Paused(reason=DONE)]
```

Notice how we get back only the first `Request` and its (empty) body, then a `Paused` event.

We process the first request:

```
In [22]: conn.send(h11.Response(status_code=200, headers=[]))
Out[22]: b'HTTP/1.1 200 \r\ntransfer-encoding: chunked\r\n\r\n'

In [23]: conn.send(h11.EndOfMessage())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[23]: b'0\r\n\r\n'
```

And then reset the connection to handle the next:

```
In [24]: conn.prepare_to_reuse()
```

This has unpaused our receive buffer, so now we're ready to read more data from the network right? Well, no– the client is done sending data, we already have all their data, so if we block waiting for more data now, then we'll be waiting forever.

That would be bad.

Instead, what we have to do after unpausing is make an explicit call to `receive_data()` with `None` as the argument, which means "I don't have any more data for you, but could you check the data you already have buffered in case there's anything else you can parse now that you couldn't before?". And once we've done this and processed the events we get back, we can continue as normal:

```
In [25]: conn.receive_data(None)
Out[25]:
[Request(method=b'GET', target=b'/1', headers=[(b'host', b'example.com')], http_version=b'1.1'),
 EndOfMessage(headers=[])]
```

It is always safe to call `conn.receive_data(None)`; if there aren't any new events to return, it will simply return `[]`, and if the connection is paused, it will return a `Paused` event. If you want to be conservative, you can defensively call this immediately before issuing any blocking read.

1.2.8 Closing connections

h11 represents a connection shutdown with the special event type *ConnectionClosed*. You can send this event, in which case *send()* will simply update the state machine and then return `None`. You can receive this event, if you call `conn.receive_data(b"")`. (The actual receipt might be delayed if the connection is *paused*.) It's safe and legal to call `conn.receive_data(b"")` multiple times, and once you've done this once, then all future calls to *receive_data()* will also return `ConnectionClosed()`:

```
In [26]: conn = h11.Connection(our_role=h11.CLIENT)

In [27]: conn.receive_data(b"")
Out[27]: [ConnectionClosed()]

In [28]: conn.receive_data(b"")
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[28]: [ConnectionClosed()]

In [29]: conn.receive_data(None)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[29]: [ConnectionClosed()]
```

(Or if you try to actually pass new data in after calling `conn.receive_data(b"")`, that will raise an exception.)

h11 is careful about interpreting connection closure in a *half-duplex fashion*. TCP sockets pretend to be a two-way connection, but really they're two one-way connections. In particular, it's possible for one party to shut down their sending connection – which causes the other side to be notified that the connection has closed via the usual `socket.recv(...) -> b""` mechanism – while still being able to read from their receiving connection. (On Unix, this is generally accomplished via the `shutdown(2)` system call.) So, for example, a client could send a request, and then close their socket for writing to indicate that they won't be sending any more requests, and then read the response. It's this kind of closure that is indicated by h11's *ConnectionClosed*: it means that this party will not be sending any more data – nothing more, nothing less. You can see this reflected in the *state machine*, in which one party transitioning to *CLOSED* doesn't immediately halt the connection, but merely prevents it from continuing for another request/response cycle.

The state machine also indicates that *ConnectionClosed* events can only happen in certain states. This isn't true, of course – any party can close their connection at any time, and h11 can't stop them. But what h11 can do is distinguish between clean and unclean closes. For example, if both sides complete a request/response cycle and then close the connection, that's a clean closure and everyone will transition to the *CLOSED* state in an orderly fashion. On the other hand, if one party suddenly closes the connection while they're in the middle of sending a chunked response body, or when they promised a `Content-Length:` of 1000 bytes but have only sent 500, then h11 knows that this is a violation of the HTTP protocol, and will raise a *ProtocolError*. Basically h11 treats an unexpected close the same way it would treat unexpected, uninterpretable data arriving – it lets you know that something has gone wrong.

As a client, the proper way to perform a single request and then close the connection is:

1. Send a *Request* with `Connection: close`
2. Send the rest of the request body
3. Read the server's *Response* and body
4. `conn.our_state is h11.MUST_CLOSE` will now be true. Call `conn.send(ConnectionClosed())` and then close the socket. Or really you could just close the socket – the thing calling `send` will do is raise an error if you're not in *MUST_CLOSE* as expected. So it's between you and your conscience and your code reviewers.

(Technically it would also be legal to shutdown your socket for writing as step 2.5, but this doesn't serve any purpose and some buggy servers might get annoyed, so it's not recommended.)

As a server, the proper way to perform a response is:

1. Send your *Response* and body

2. Check if `conn.our_state is h11.MUST_CLOSE`. This might happen for a variety of reasons; for example, if the response had unknown length and the client speaks only HTTP/1.0, then the client will not consider the connection complete until we issue a close.

You should be particularly careful to take into consideration the following note fromx RFC 7230 section 6.6:

> If a server performs an immediate close of a TCP connection, there is a significant risk that the client will not be able to read the last HTTP response. If the server receives additional data from the client on a fully closed connection, such as another request that was sent by the client before receiving the server's response, the server's TCP stack will send a reset packet to the client; unfortunately, the reset packet might erase the client's unacknowledged input buffers before they can be read and interpreted by the client's HTTP parser.
>
> To avoid the TCP reset problem, servers typically close a connection in stages. First, the server performs a half-close by closing only the write side of the read/write connection. The server then continues to read from the connection until it receives a corresponding close by the client, or until the server is reasonably certain that its own TCP stack has received the client's acknowledgement of the packet(s) containing the server's last response. Finally, the server fully closes the connection.

1.2.9 Switching protocols

h11 supports two kinds of "protocol switches": requests with method `CONNECT`, and the newer `Upgrade:` header, most commonly used for negotiating WebSocket connections. Both follow the same pattern: the client proposes that they switch from regular HTTP to some other kind of interaction, and then the server either rejects the suggestion – in which case we return to regular HTTP rules – or else accepts it. (For `CONNECT`, acceptance means a response with 2xx status code; for `Upgrade:`, acceptance means an *`InformationalResponse`* with status `101 Switching Protocols`) If the proposal is accepted, then both sides switch to doing something else with their socket, and h11's job is done.

As a developer using h11, it's your responsibility to send and interpret the actual `CONNECT` or `Upgrade:` request and response, and to figure out what to do after the handover; it's h11's job to understand what's going on, and help you make the handover smoothly.

Specifically, what h11 does is *pause* parsing incoming data at the boundary between the two protocols, and then you can retrieve any unprocessed data from the *`Connection.trailing_data`* attribute.

1.2.10 Support for `sendfile()`

Many networking APIs provide some efficient way to send particular data, e.g. asking the operating system to stream files directly off of the disk and into a socket without passing through userspace.

It's possible to use these APIs together with h11. The basic strategy is:

- Create some placeholder object representing the special data, that your networking code knows how to "send" by invoking whatever the appropriate underlying APIs are.
- Make sure your placeholder object implements a `__len__` method returning its size in bytes.
- Call `conn.send_with_data_passthrough(Data(data=<your placeholder object>))`
- This returns a list whose contents are a mixture of (a) bytes-like objects, and (b) your placeholder object. You should send them to the network in order.

Here's a sketch of what this might look like:

```
class FilePlaceholder:
    def __init__(self, file, offset, count):
        self.file = file
```

```
        self.offset = offset
        self.count = count

    def __len__(self):
        return self.count

def send_data(sock, data):
    if isinstance(data, FilePlaceholder):
        # socket.sendfile added in Python 3.5
        sock.sendfile(data.file, data.offset, data.count)
    else:
        sock.sendfile(data)

placeholder = FilePlaceholder(open("...", "rb"), 0, 200)
for data in conn.send_with_data_passthrough(Data(data=placeholder)):
    send_data(sock, data)
```

This works with all the different framing modes (`Content-Length`, `Transfer-Encoding: chunked`, etc.) – h11 will add any necessary framing data, update its internal state, and away you go.

1.2.11 Identifying h11 in requests and responses

According to RFC 7231, client requests are supposed to include a `User-Agent:` header identifying what software they're using, and servers are supposed to respond with a `Server:` header doing the same. h11 doesn't construct these headers for you, but to make it easier for you to construct this header, it provides:

`h11.`**`PRODUCT_ID`**

A string suitable for identifying the current version of h11 in a `User-Agent:` or `Server:` header.

The version of h11 that was used to build these docs identified itself as:

```
In [30]: h11.PRODUCT_ID
Out[30]: 'h11/0.5.0'
```

1.3 Details of our HTTP support for HTTP nerds

h11 only speaks HTTP/1.1. It can talk to HTTP/1.0 clients and servers, but it itself only does HTTP/1.1.

We fully support HTTP/1.1 keep-alive.

We have a little bit of support for HTTP/1.1 pipelining – basically the minimum that's required by the standard. In server mode we can handle pipelined requests in a serial manner, responding completely to each request before reading the next (and our API is designed to make it easy for servers to keep this straight). Client mode doesn't support pipelining at all. As far as I can tell, this matches the state of the art in all the major HTTP implementations: the consensus seems to be that HTTP/1.1 pipelining was a nice try but unworkable in practice, and if you really need pipelining to work then instead of trying to fix HTTP/1.1 you should switch to HTTP/2.0.

The HTTP/1.0 `Connection: keep-alive` pseudo-standard is currently not supported. (Note that this only affects h11 as a server, because h11 as a client always speaks HTTP/1.1.) Supporting this would be possible, but it's fragile and finicky and I'm suspicious that if we leave it out then no-one will notice or care. HTTP/1.1 is now almost old enough to vote in United States elections. I get that people sometimes write HTTP/1.0 clients because they don't want to deal with annoying stuff like chunked encoding, and I completely sympathize with that, but I'm guessing that you're not going to find too many people these days who care desperately about keep-alive *and at the same time* are too lazy to implement Transfer-Encoding: chunked. Still, this would be my bet as to the missing feature that people are most likely to eventually complain about...

Of the headers defined in RFC 7230, the ones h11 knows and has some special-case logic to care about are: `Connection:`, `Transfer-Encoding:`, `Content-Length:`, `Host:`, `Upgrade:`, and `Expect:` (which is really from RFC 7231 but whatever). The other headers in RFC 7230 are `TE:`, `Trailer:`, and `Via:`; h11 also supports these in the sense that it ignores them and that's really all it should be doing.

Transfer-Encoding support: we only know `chunked`, not `gzip` or `deflate`. We're in good company in this: node.js at least doesn't handle anything besides `chunked` either. So I'm not too worried about this being a problem in practice. But I'm not majorly opposed to adding support for more features here either.

A quirk in our *`Response`* encoding: we don't bother including ascii status messages – instead of `200 OK` we just say `200`. This is totally legal and no program should care, and it lets us skip carrying around a pointless table of status message strings, but I suppose it might be worth fixing at some point.

When parsing chunked encoding, we parse but discard "chunk extensions". This is an extremely obscure feature that allows arbitrary metadata to be interleaved into a chunked transfer stream. This metadata has no standard uses, and proxies are allowed to strip it out. I don't think anyone will notice this lack, but it could be added if someone really wants it; I just ran out of energy for implementing weirdo features no-one uses.

Currently we *do* implement support for "obsolete line folding" when reading HTTP headers. This is an optional part of the spec – conforming HTTP/1.1 implementations MUST NOT send continuation lines, and conforming HTTP/1.1 servers MAY send 400 Bad Request responses back at clients who do send them (ref). I'm tempted to remove this support, since it adds some complicated and ugly code right at the center of the request/response parsing loop, and I'm not sure whether anyone actually needs it. Unfortunately a few major implementations that I spot-checked (node.js, go) do still seem to support reading such headers (but not generating them), so it might or might not be obsolete in practice – it's hard to know.

1.4 History of changes

1.4.1 v0.5.0

- Initial release.

h